Tampereen yliopisto
Tampere University

TIE-02306 ItSE

Introduction to Software Engineering

5 credit units

02-general-ItSE-2019-v4

Tampereen yliopisto
Tampere University

Course contents (plan)

1. Course basics, intro
2. **Sw Eng in general, overview**
3. Requirements
4. Different software systems
5. Basic UML Diagrams (”Class”, Use Case, Navigation)
6. Life Cycle models
7. UML diagrams, in more detail
8. Quality and Testing
9. Project work
10. Project management
11. Open source, APIs, IPR
12. Embedded systems
13. Recap

Tampereen yliopisto
Tampere University

2. Sw Eng in general, overview

- Life cycles, S(D)LC
- software engineering vs. ”other” work…
- sw eng IS teamwork
-
- how to get sw; buy (build), tailor/modify, COTS, SaaS,…
-
- it MAY be worth thinking to modify your way of work, not the software…
-
- ethics (Code of Conduct)
-
- documentation
- reuse

Tampereen yliopisto
Tampere University

First, general course matters

- 39 signed students at POP/ROCK
- 8 full (4/4) project groups, and one 3/4 -> 9 groups

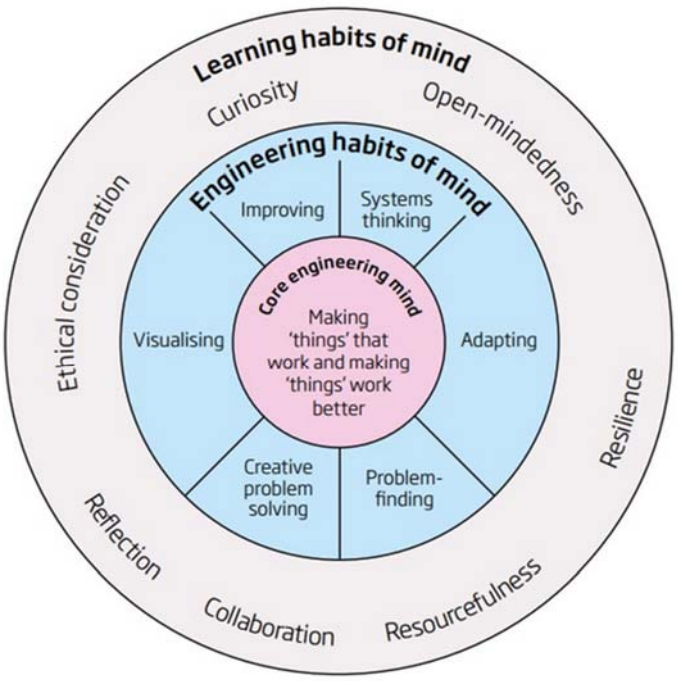
Ulla: groups 1, 3, 5, 7, 9

Valentina: groups 2, 4, 6, 8.

- Trello board and Product Backlog ready at the end of week
- invite all course personnel to your Trello board

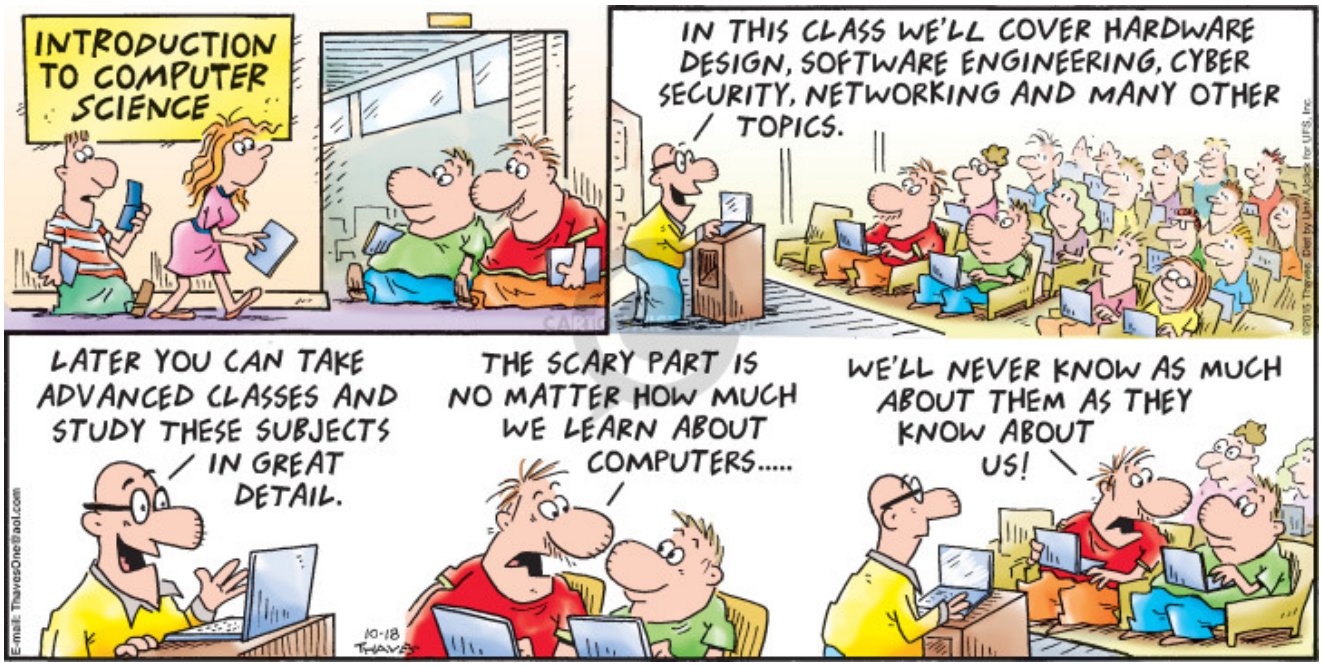
- WE2 will be WED and THU this week
- **WE2 is about Trello, take computers/tablets with you**

- **Ulla and Valentina are travelling week 38, WE3 by Tensu.**

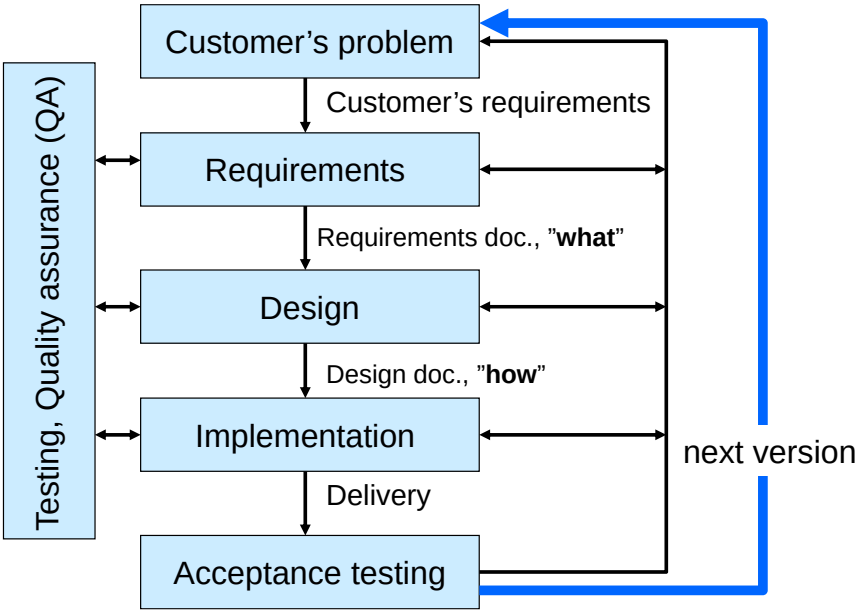


Engineering habits of mind (EHoM) and wider learning habits of mind (LHoM) (Credit: Royal Academy of Engineering)

Watch YouTube video; *"amazing mind reader reveals his gift"*

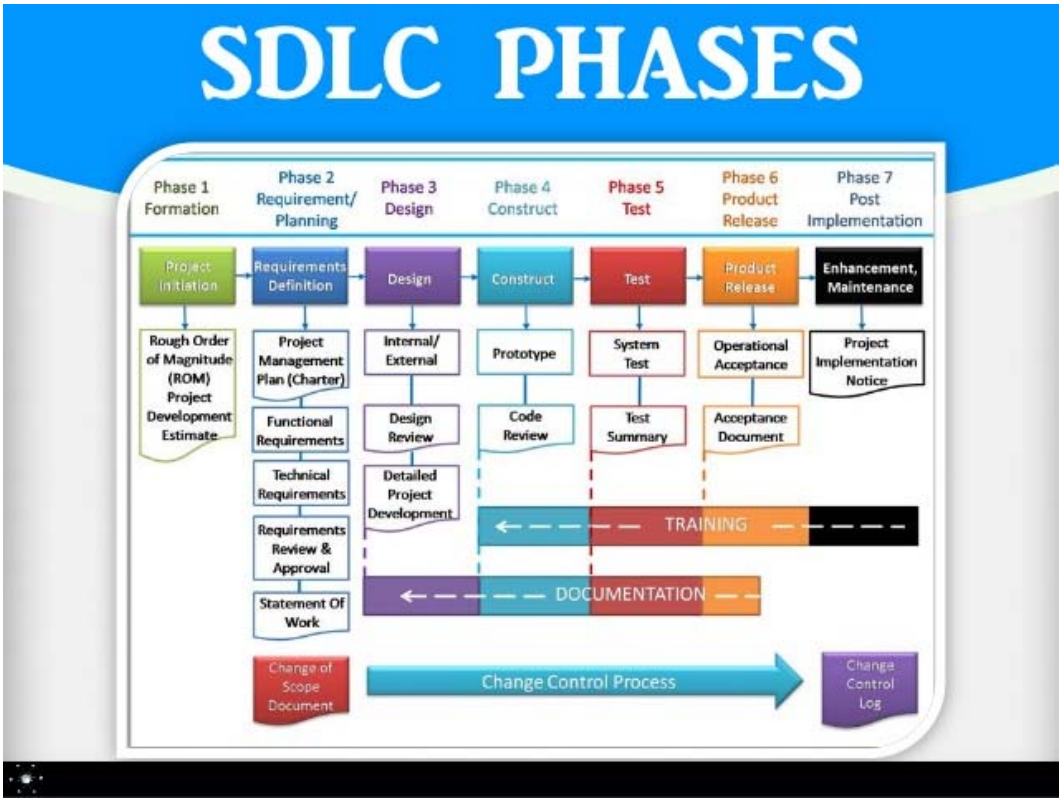


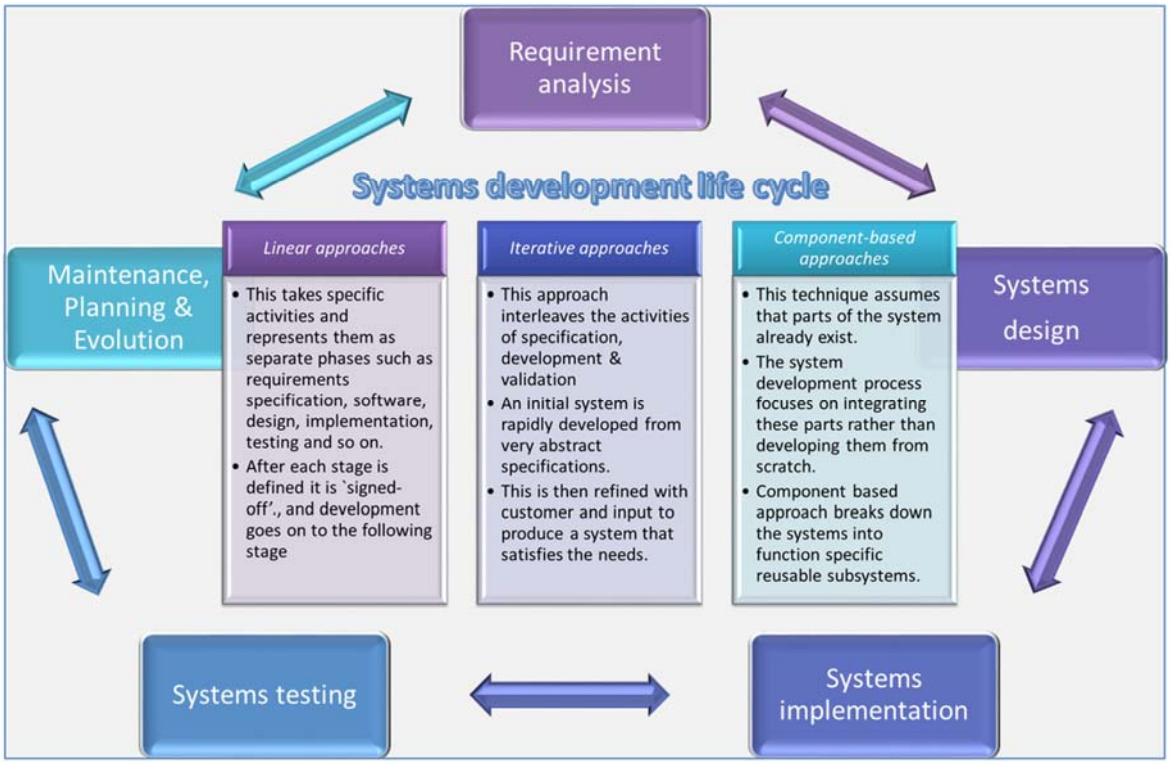
Development process; there may be many variations (paths)



Iterations are done if necessary.

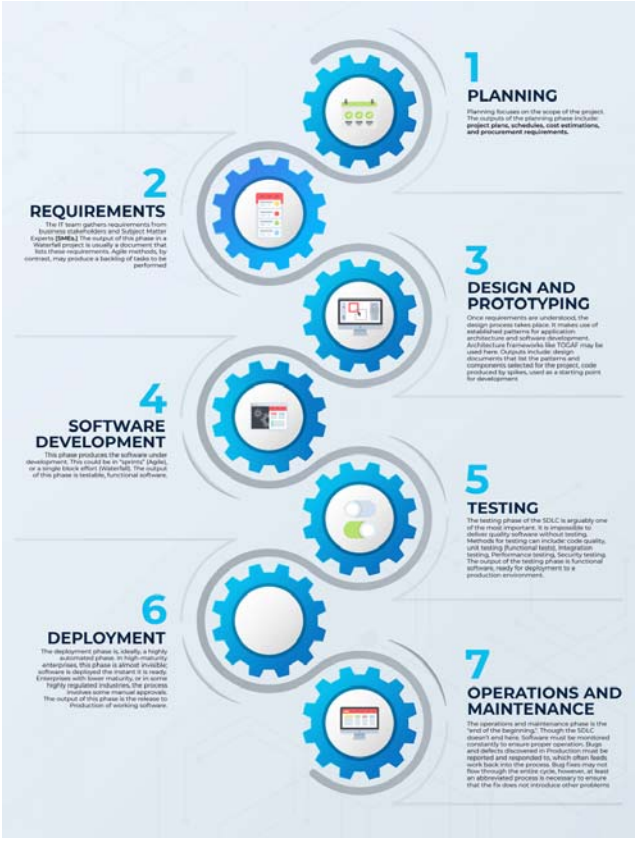
Tampereen yliopisto
Tampere University

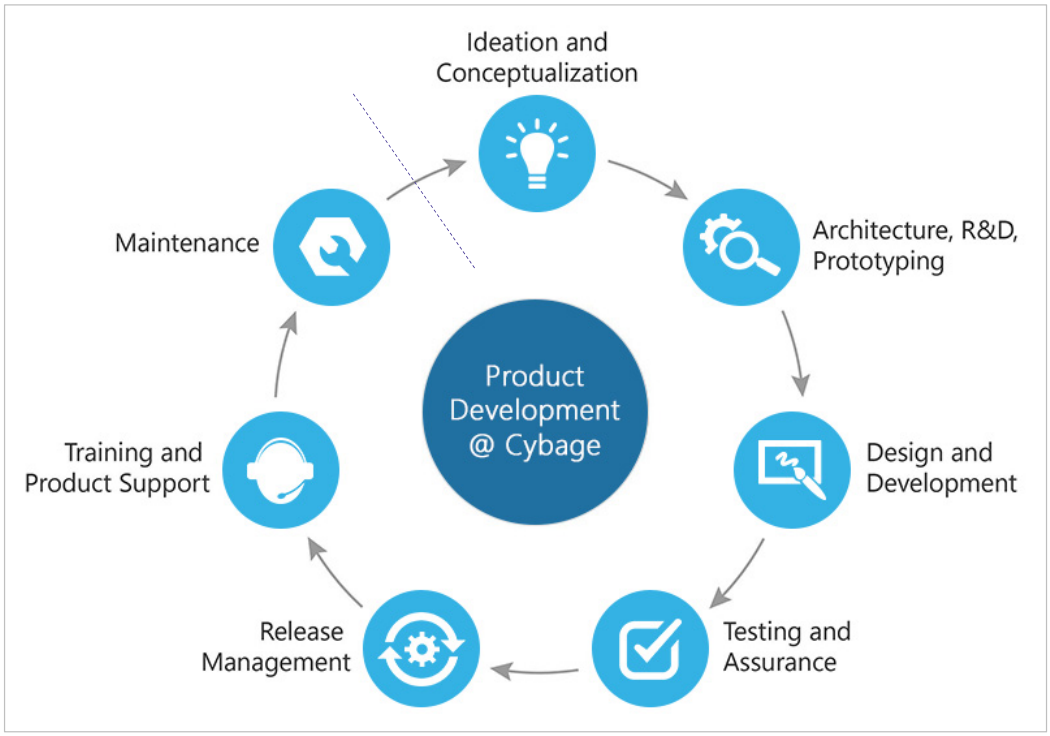

SDLC PHASES
Phase 1 Formation
Phase 2 Requirement/ Planning
Phase 3 Design
Phase 4 Construct
Phase 5 Test
Phase 6 Product Release
Phase 7 Post Implementation
Project Initiation
Requirements Definition
Design
Construct
Test
Product Release
Enhancement, Maintenance
Rough Order of Magnitude (ROM) Project Development Estimate
Project Management Plan (Charter)
Functional Requirements
Technical Requirements
Requirements Review & Approval
Statement Of Work
Internal/ External
Design Review
Detailed Project Development
Prototype
Code Review
System Test
Test Summary
Operational Acceptance
Acceptance Document
Project Implementation Notice
TRAINING
DOCUMENTATION
Change of Scope Document
Change Control Process
Change Control Log


Requirement analysis
Systems development life cycle
Maintenance, Planning & Evolution
Linear approaches
• This takes specific activities and represents them as separate phases such as requirements specification, software, design, implementation, testing and so on.
• After each stage is defined it is 'signed-off'., and development goes on to the following stage
Iterative approaches
• This approach interleaves the activities of specification, development & validation
• An initial system is rapidly developed from very abstract specifications.
• This is then refined with customer and input to produce a system that satisfies the needs.
Component-based approaches
• This technique assumes that parts of the system already exist.
• The system development process focuses on integrating these parts rather than developing them from scratch.
• Component based approach breaks down the systems into function specific reusable subsystems.
Systems design
Systems testing
Systems implementation


7 OF THE SOFTWARE
DEVELOPMENT LIFE CYCLE
PHASES
RAYGUN

BROUGHT TO YOU BY
RAYGUN
www.raygun.com


1
PLANNING
2
REQUIREMENTS
3
DESIGN AND PROTOTYPING
4
SOFTWARE DEVELOPMENT
5
TESTING
6
DEPLOYMENT
7
OPERATIONS AND MAINTENANCE


Ideation and Conceptualization
Architecture, R&D, Prototyping
Design and Development
Testing and Assurance
Release Management
Training and Product Support
Maintenance
Product Development @ Cybage

software (system) projects

Software project may be

- building a **new** system from scratch
- **enhancement** of an existing system (e.g. adding features, refactoring)
- **re-development**, re-designing, replacing an existing system (e.g. transferring system to a new platform, and update architecture as a side effect).

Software system may be described by its size (not very accurate), 1 / 2

Tiny - 1000 lines or less. Easy to keep every tiny detail and every line in your head and know exactly where it is.

Small - 1000-5000 lines. You know the details of individual modules and functions and within a line range of where to locate a particular set of functionality with ease. You still have deep knowledge of the bulk of the code.

Medium - 5000 to 20000 lines. You know the structure of the entire project in your head and know roughly what module a particular function exists in and how far in to scroll to find it. You're only maintaining details of code you're actively working at this point.

Large - 20000 to 50000 lines. Detail knowledge is limited to immediate work and everything else is having a birds-eye sense of where stuff is in the code. You're working off high-level structural knowledge and will start using tools to pinpoint things you've forgotten the exact location for. This tends to also be the mental threshold of where an experienced programmer can hold the picture of the entire codebase in their head.

BTW; how many lines should be for testing or code comments ?

[Matt Pickering, 30+ yrs prof dev]

Software system may be described by its size (not very accurate), 2 / 2

Very Large - 50000 to 250000 lines. You're at multiple team members at this point and you're treating the codebase as a set of interlocked projects. Coordination for changes is absolutely essential at this point. Only a handful of developers could maintain a complete mental picture of the codebase at this point and would be deep specialists in it having worked on it for years.

Huge - 250000 to 1M lines. Substantial large scale software engineering efforts. Projects are now an interlocked set of loosely or tightly coupled integration points across multiple applications. No individual can generally hold the detail of the project beyond the block diagram level. Blocks on the diagram are very large software efforts in their own right. Too large for anyone to hold in their head as whole at any level of detail.

Massive - Anything north of 1M lines. These are multi-year, engineering discipline level efforts where documentation needed to coordinate and manage change outstrips the software development effort. Operating systems, large scale enterprise products, specialty software code bases that handle mission critical, large scale functionality or high volumes and so on. These code bases tend to have grown to this size and have evolved over years of work. Impossible for anyone to know fully.

[Matt Pickering, 30+ yrs prof dev]

Example: VFS for Git (ex: GVFS)

This project was formerly known as **GVFS** (**Git Virtual File System**). It is undergoing a rename to VFS for Git.

VFS stands for Virtual File System. VFS for Git virtualizes the file system beneath your git repo so that git and all tools see what appears to be a normal repo, but VFS for Git only downloads objects as they are needed. VFS for Git also manages the files that git will consider, to ensure that git operations like status, checkout, etc., can be as quick as possible because they will only consider the files that the user has accessed, not all files in the repo.

The Windows code base is approximately 3.5M files and, when checked in to a Git repo, results in a repo of about 300GB. Further, the Windows team is about 4,000 engineers and the engineering system produces 1,760 daily “lab builds” across 440 branches in addition to thousands of pull request validation builds.

https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/

https://github.com/microsoft/VFSForGit

Example, about Windows repositories

Small repos: Even though GVFS is a product meant for the world’s largest repos, GVFS itself has a rather ordinary codebase when it comes to its scale needs. GVFS stats:

Repo size: 10MB packfile, 2MB working directory, 400 files

Team: 10 people

Branches: 300

Build/Test: 30s build, 10s unit tests, 30m functional tests.

Extra large repos: The Windows codebase is in a class of its own. This code includes all of OneCore, Desktop, Server, Xbox, IOT, Mobile, and HoloLens. Along with the fact that this codebase has been around for decades and still supports legacy features all the way back to the beginning of Windows, you can imagine that the codebase is quite large. Windows stats:

Repo size: 100GB packfile, 300GB working directory, 3.5M files

Team: 4000 people

Branches: Estimated to reach between 150K to 250K

Build/Test: A few seconds to few minutes incremental build, 12h full build, a few minutes incremental tests, several hours for full validation.

https://docs.microsoft.com/en-us/azure/devops/learn/git/git-at-scale#extra-large-repos

”Average amount” of bugs

Folklore: ”industry average” amount is 15..50 bugs in KLOC.

Microsoft applications; 10..20 defects / KLOC during in-house testing, 0,5 defects / KLOC in released code. [2007]

Folklore: ”A C++ coder will be productive after 1,5 years of working.”

Project size (number of lines of code)	Average error density
less than 2K	0 - 25 errors per 1000 lines of code
2K - 16K	0 - 40 errors per 1000 lines of code
16K - 64K	0.5 - 50 errors per 1000 lines of code
64K - 512K	2 - 70 errors per 1000 lines of code
512K and more	4 - 100 errors per 1000 lines of code

https://hownot2code.com/2017/11/08/search-for-bugs-in-code-at-the-early-stage/

Some lines-of-code (LOC) counts

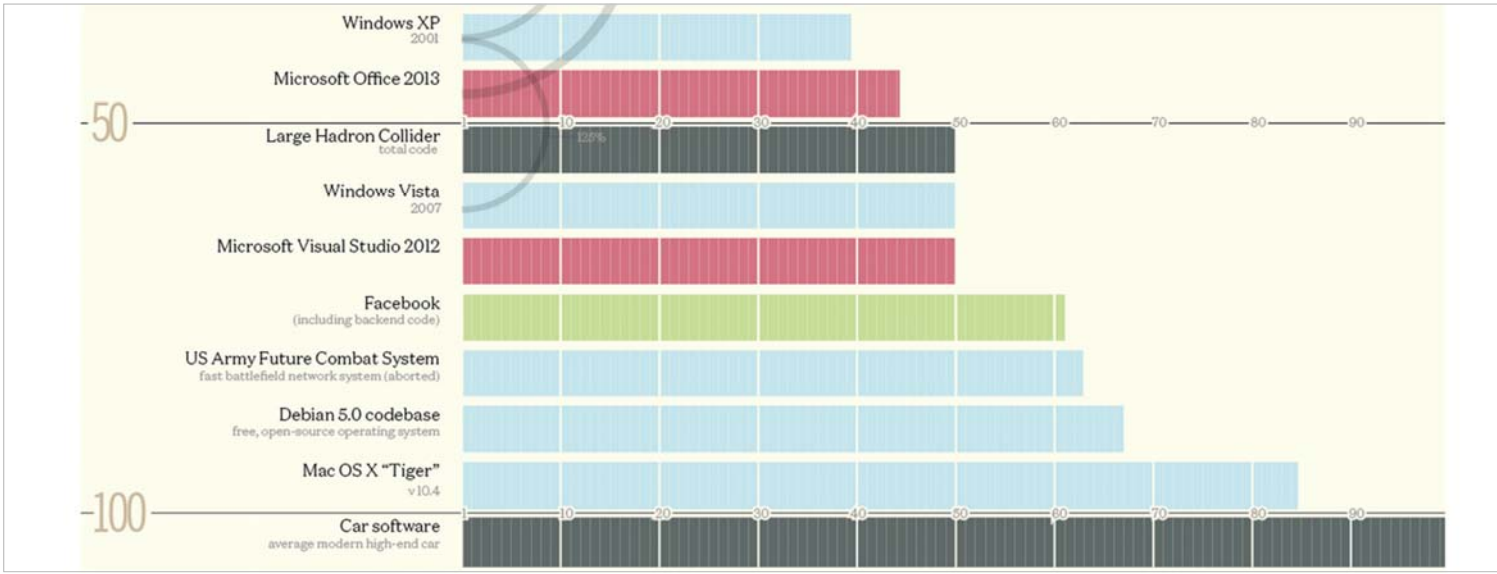
It is difficult to get exact answers, but it is estimated that

-
- Windows XP: 40 MLOC
- Windows Vista: 50 MLOC
- Windows 7: 40 MLOC
- Windows 8: 50..60 MLOC
- Windows 10: ??
- Google; more than 2 BLOC, 9 M source files, 25000 developers

Is only one defect per 1000 lines of code enough for good quality ?

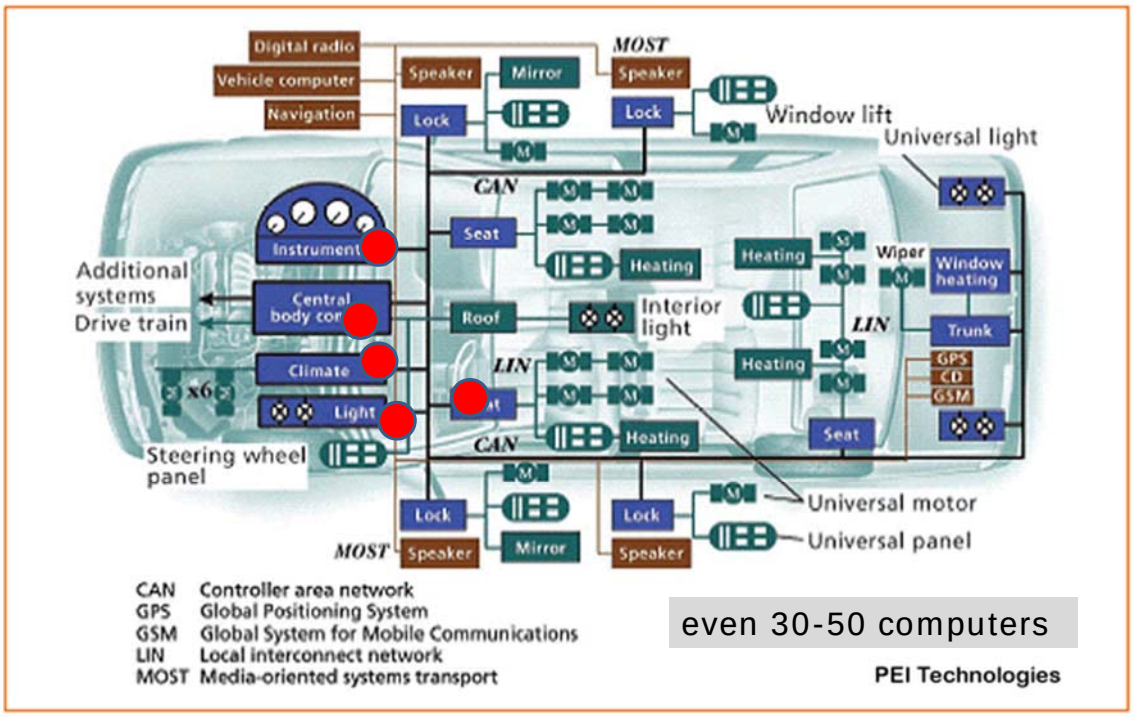
- F-22 fighter; 1,7 MLOC
- Boeing 787; 14 MLOC
- F-35 fighter; 24 MLOC.

Some lines of code (LOC) amounts

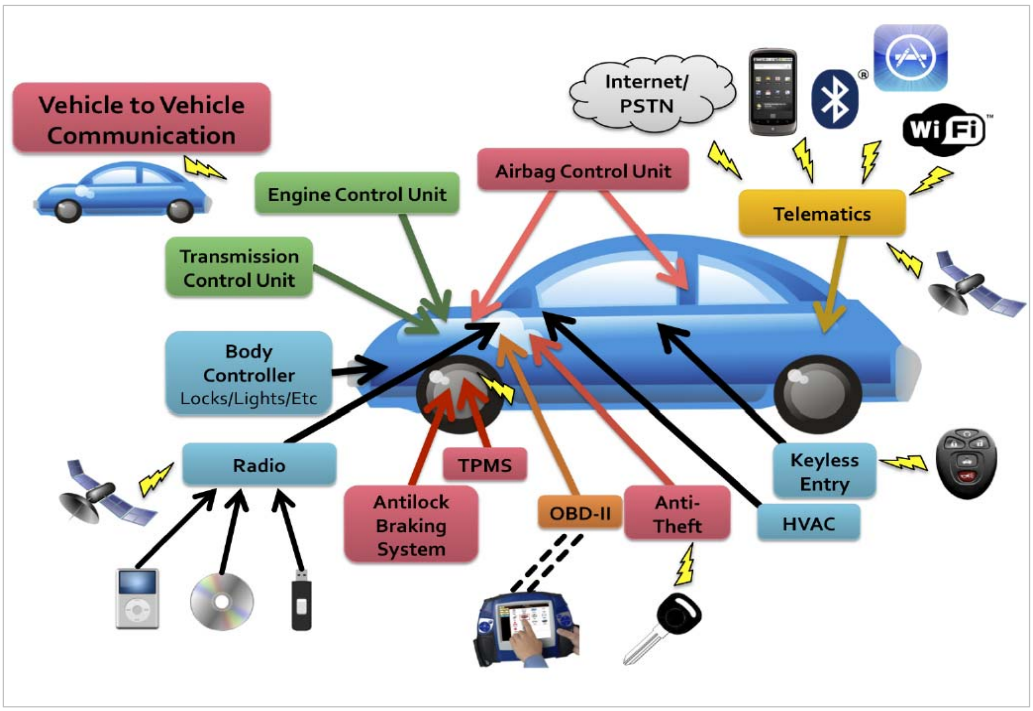


What is this… a game ?

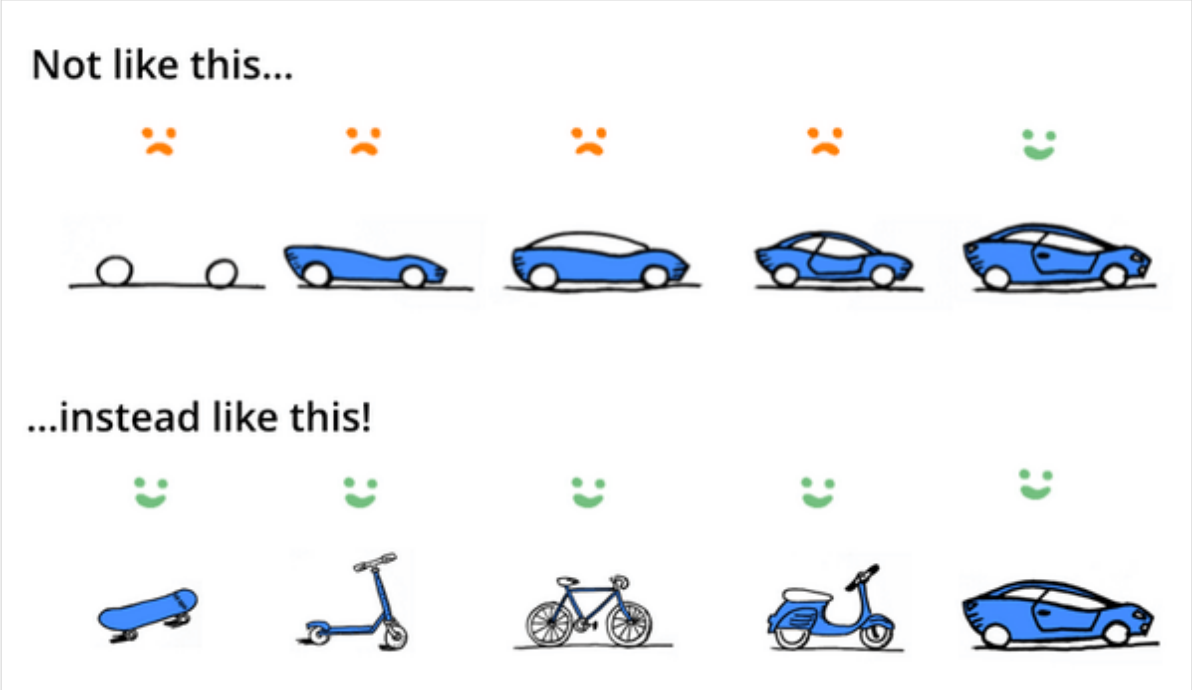
http://www.aa1car.com/library/can_systems.htm



http://www.dailytech.com/Charlie+Miller+Releases+Open+Source+Car+Sabotage+Toolkit/article33308.htm

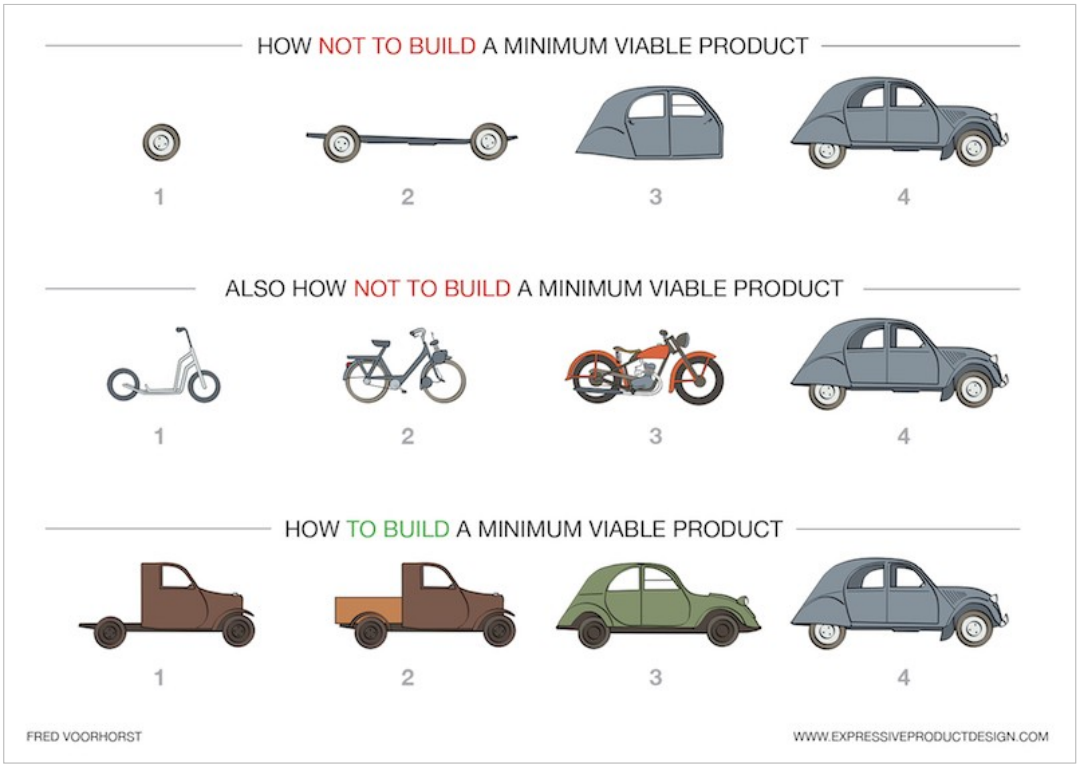


Iterative approach



Minumum Viable Product (MVP) is NOT the ”quickest hack” you dare to send to customer. This is a common misunderstanding.

MVP is the quickest made product version which brings profit/value to vendor and customer.



Folklore: sw developer's working hour "cost" inside a company may be 70..100 euros.

So… a one-person project lasting six months… equals half a person-year. 1000 working hours yearly ? That makes 500 h x 70 e = 35000 euros.

10 person project, six months, 350000 euros.

Project is much more than just coding…

Plus 25 % profit to vendor ?

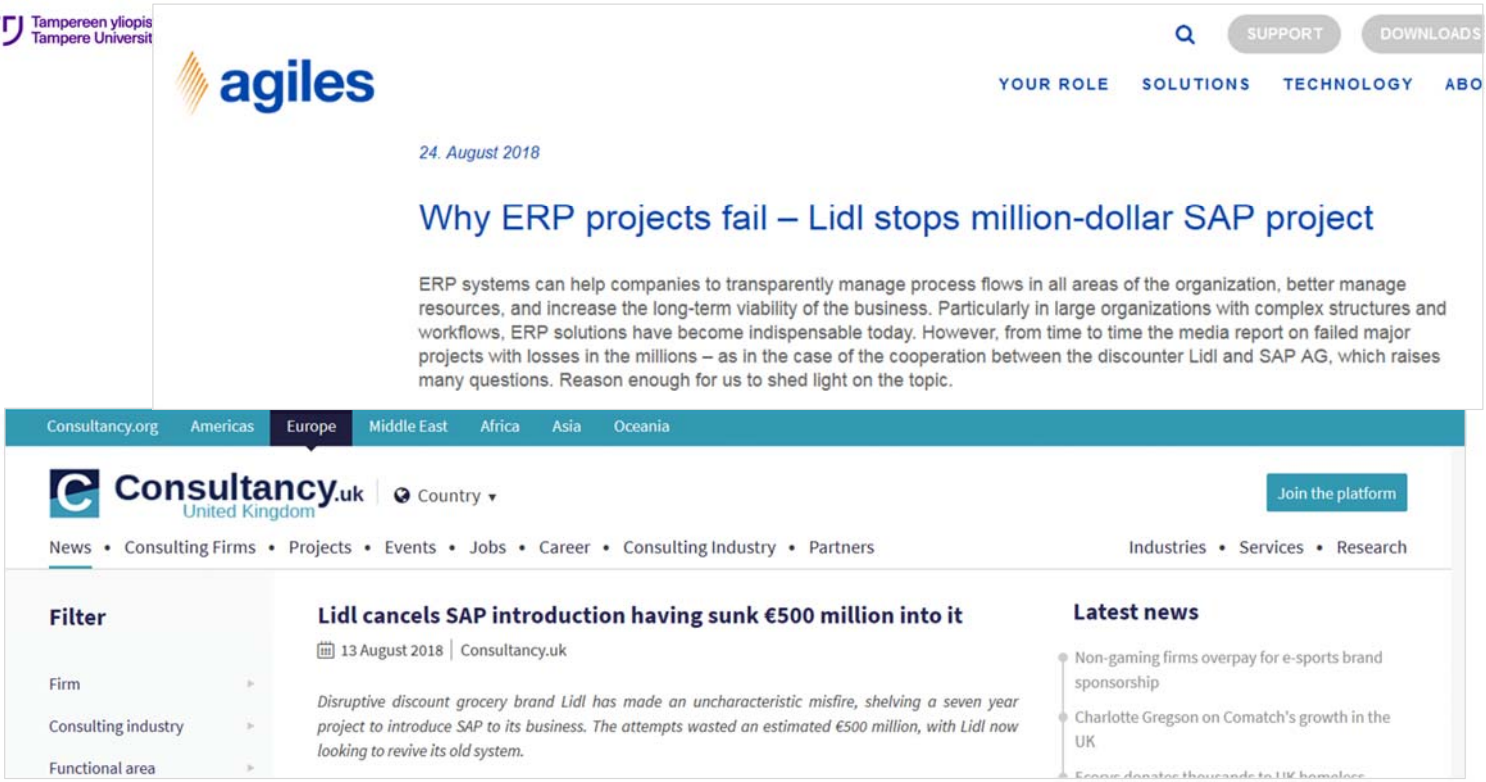
And then somebody wonders why software is expensive ??

Senior consultant may take 500..1000 euros / day.

COMPARISON OF SOFTWARE DEVELOPMENT COMPANIES

	Enterprise	Mid-size company	Small team	Freelancers
Size (number of people)	Over 1000	50-1000	Below 50	Individuals
Technologies and skillsets available	Any	Any	Limited	Very limited
On-demand availability	You can hire any specialist (designer, QA, marketer) on a part-time basis	You can hire any specialist (designer, QA, marketer) on a part-time basis	If a specific skill is not available in the team, you have to find a new team	You have to recruit each new specialist individually
Scale up / scale down option	Can easily meet your growing requirements or cut team to the size you need	Can easily meet your growing requirements or cut team to the size you need	Have insufficient resources to keep up with your growth	Never ending process of finding and hiring new freelancers
Administrative issues and overheads (maintenance, sick leaves, hardware requirements)	Covered	Covered	Partly / Uncovered	Uncovered
Control over the development process	Developed project management practices, transparent processes, reports and monitoring	Developed project management practices, transparent processes, reports and monitoring	Very much depends on the team and management	None
Working process organization	Established, conservative and can't be changed	Flexible and can be adjusted for your needs and terms	Poorly organized	None
Customized approach	None	Provided	Provided	Provided
Risks	Low and covered	Low and covered	Medium and uncovered	Extremely high - russian roulette

www.easternpeak.com



LIDL project problem;

- want to tailor ERP system to own process
- tailoring is expensive
- tailoring is difficult, PERHAPS company would have better to change work process to fit the (well working) software ??

Read more e.g.:

http://www.bestpracticegroup.com/lidl-cancels-e500m-sap-it-project-4-learnings-to-consider/

Tampereen yliopisto
Tampere University

Software engineering is abstract, 1

Compared to ”traditional” engineering, building software is not concrete.

You can not ”see” it to estimate if it is ready and complete.

You can have some idea about the readiness only by testing, and testing a lot.

It may look good (GUI), but it may have some functionalities not implemented at all.

Two weeks before deadline, you may find one glitch/bug in user interface. The fixing may take three weeks, project gets late two weeks.

To add one simple feature to a product may take two weeks (or even more).

At halfway in a six-month sw project, you may figure out that architecture (basic structure) will not work with planned implementation, so you have to build the software again from scratch.

BTW, if you change tools or libraries during a project… you are unlucky, oh boy.

Tampereen yliopisto
Tampere University

Software engineering is abstract, 2

In a house building project, you can concretely see how the walls and floors rise up. By just watching the construction site, you can figure out how the house is being built, and you may estimate how long it may take to finish the house. You may watch inside the window and see if the rooms inside are ready or not.

But in software engineering you can not figure out the readiness state of a program just by looking the GUI demo, or code. Some minor feature might take many person-weeks to complete, on the other hand some major change to GUI may just take an hour, you newer know.

Some small change may be so difficult to make, that it is better to leave it out from the project.

E.g. at car production (line) it is easy to coordinate work, and results are visible

- press shop
- weld shop
- paint shop
- engine shop and TransAxle
- trim/final fitment.

http://www.team-bhp.com/forum/indian-car-scene/83316-automotive-manufacturing-overview.html

SLC = software life cycle

software life cycle. The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. *Note:* These phases may overlap or be performed iteratively. *Contrast with:* **software development cycle.**

TIE-02306 ItSE is about that small part

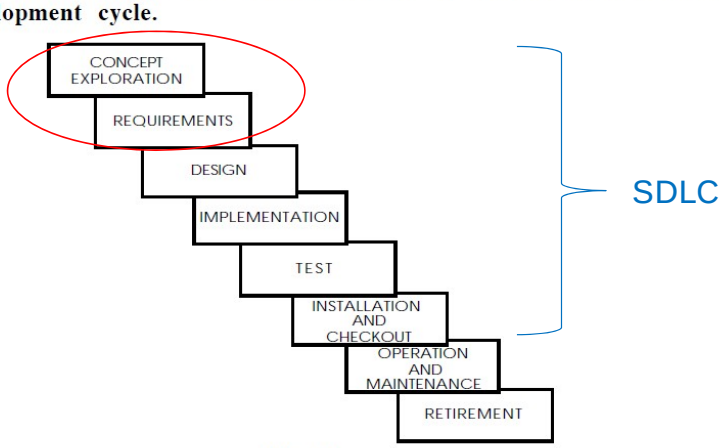


Fig 15
Sample Software Life Cycle

[**IEEE Standard Glossary of Software Engineering Terminology,** Std 610.12-1990(R2002)]

ISO/IEC/IEEE 15288:2015 Systems and software engineering – system life cycle processes

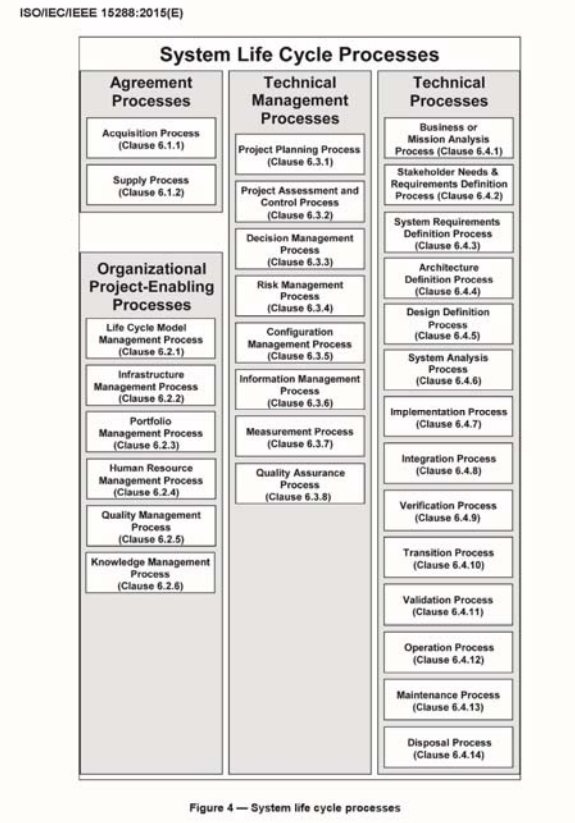


Figure 4 — System life cycle processes

ISO/IEC/IEEE 15288:2015 Systems and software engineering – system life cycle processes

TIE-02306 matters are just a small part at the beginning of a life cycle.

SRS = Software Requirements Specification

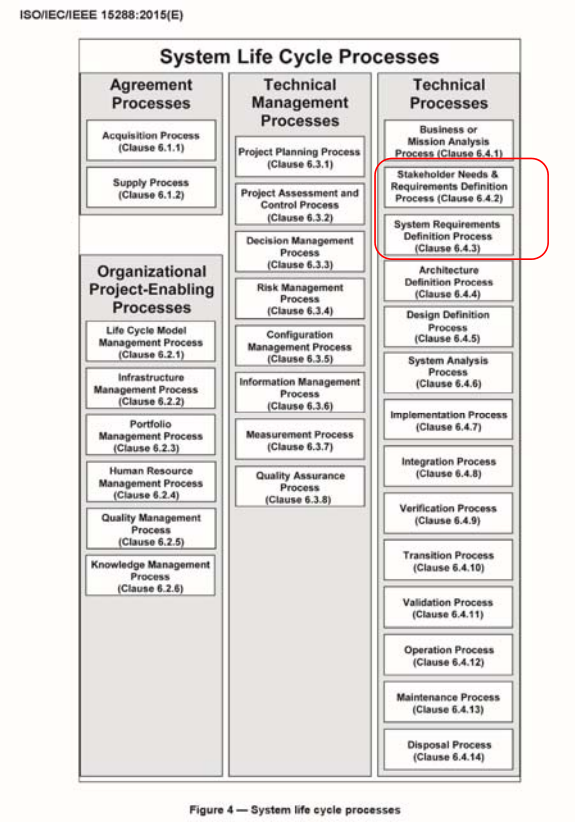
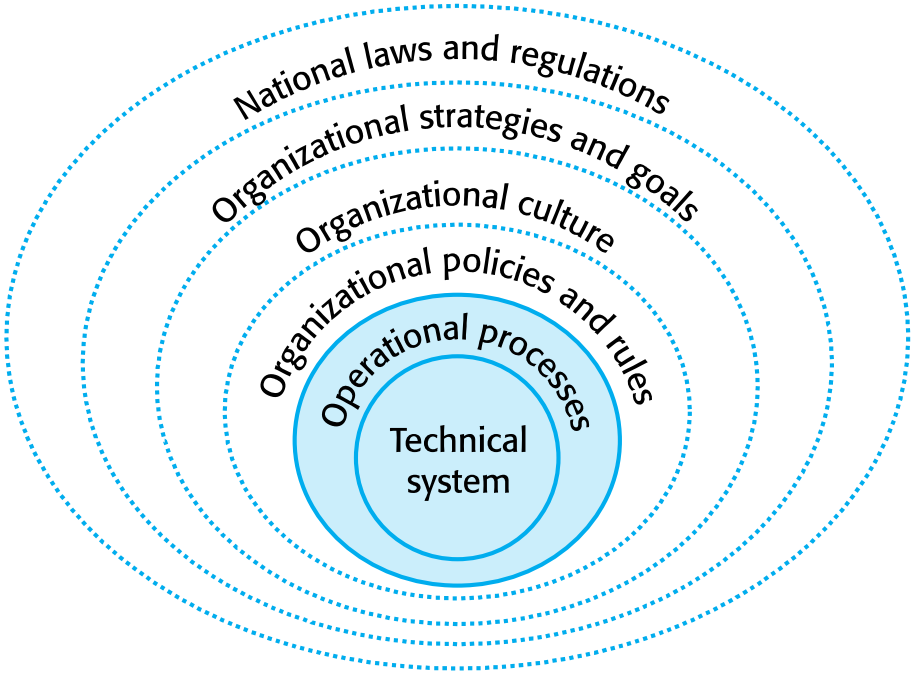
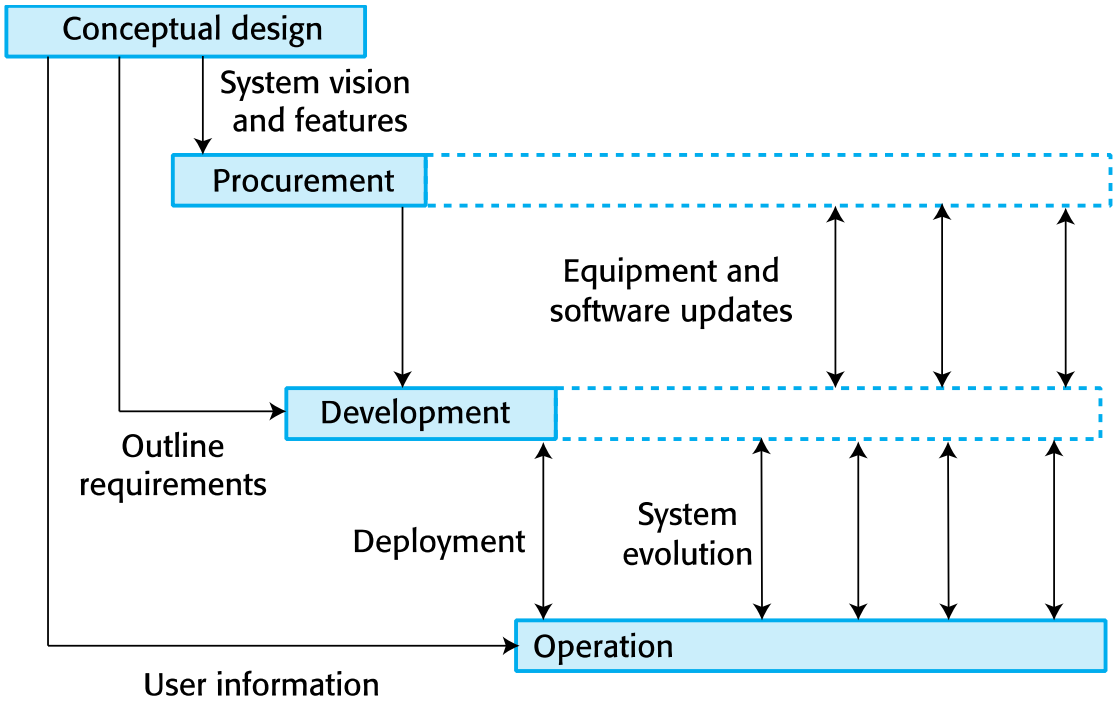


Figure 4 — System life cycle processes

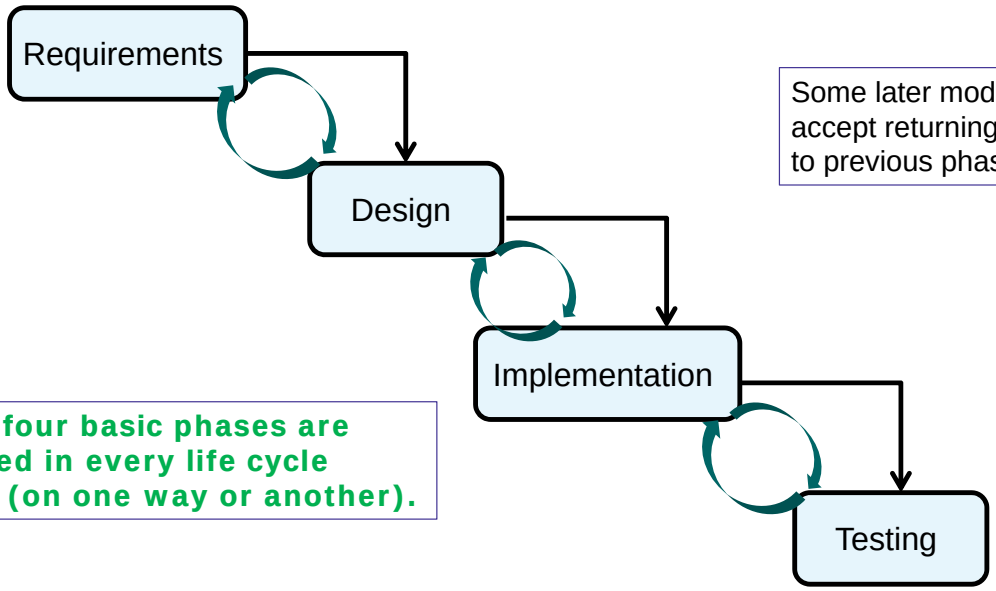
System in general [Sommerville SE10, 2015]



System life cycle [Sommerville SE10, 2015]



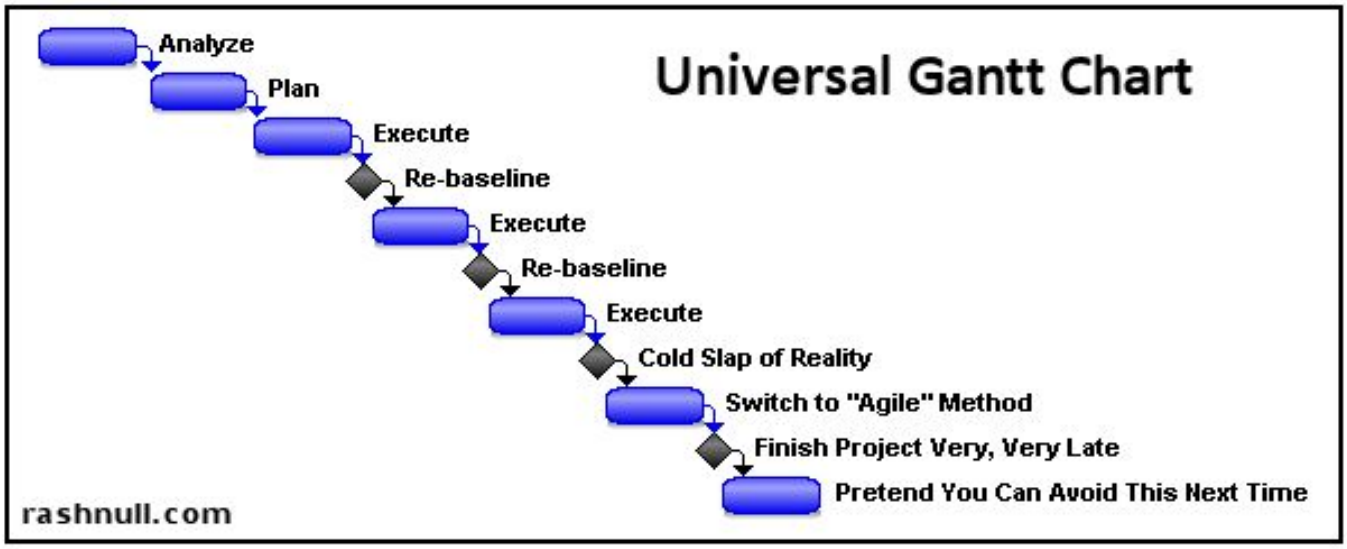
Metaphor of the waterfall



Some later models accept returning back to previous phase.

These four basic phases are included in every life cycle model (on one way or another).

Universal Gantt Chart

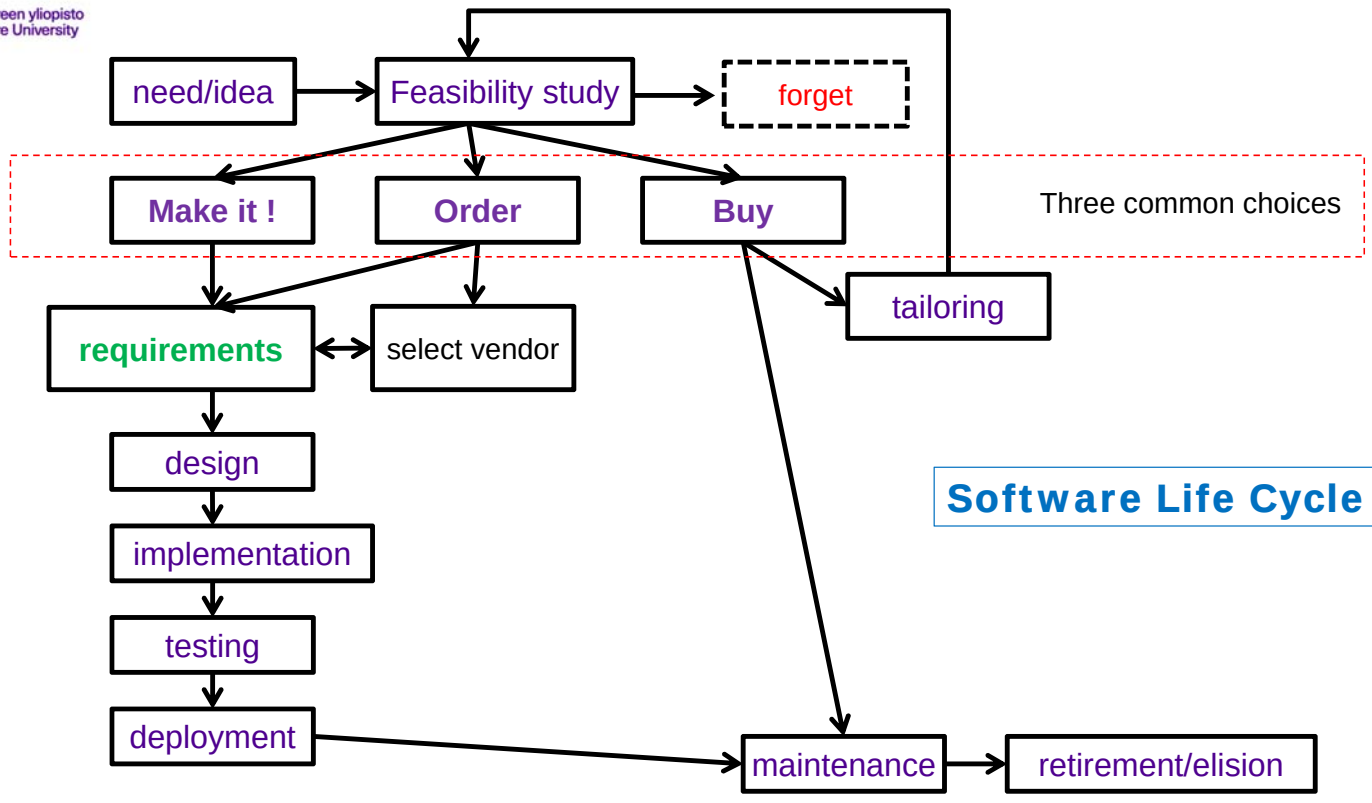


Today's ("hype" ?) software development method(ologie)s

- agile methods (agile, FI: ketterä); iterative
- **Scrum** (one agile method)
- **Kanban**; visible board, WIP (pull), optimisation
- **Lean** = eliminate waste, optimise
- **DevOps** = DevelopmentOperations (CI, CD)
- **SEMAT** = Software Engineering Method and Theory
- **SAFe** = Scaled Agile Framework, for BIG projects
- **Blended Agile** = combination of two or more Agile-methods
- **Hybrid Agile** = Agile + non-Agile-method.

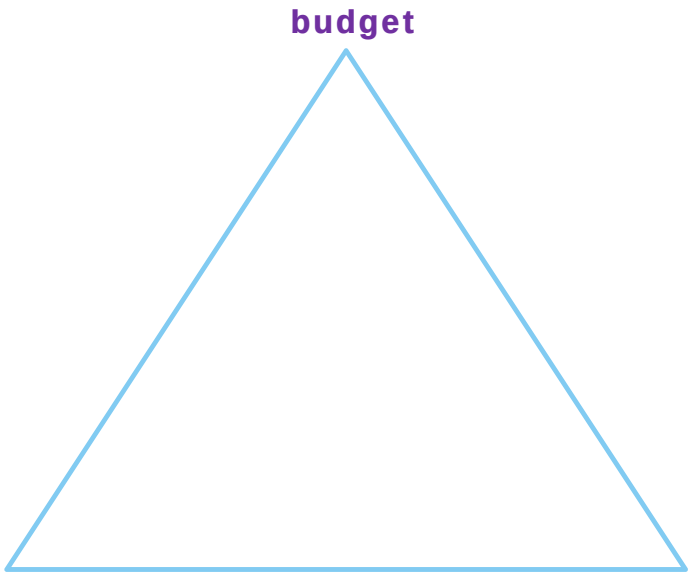
The Best Method is the one, which works best for your project.

Methodology is a set on methods (and practices). Keep your eyes and ears open for methods worth trying in your own work.



Software Life Cycle

”Iron triangle” of SW projects



Triangle is always some kind of a compromised balance between the three factors.

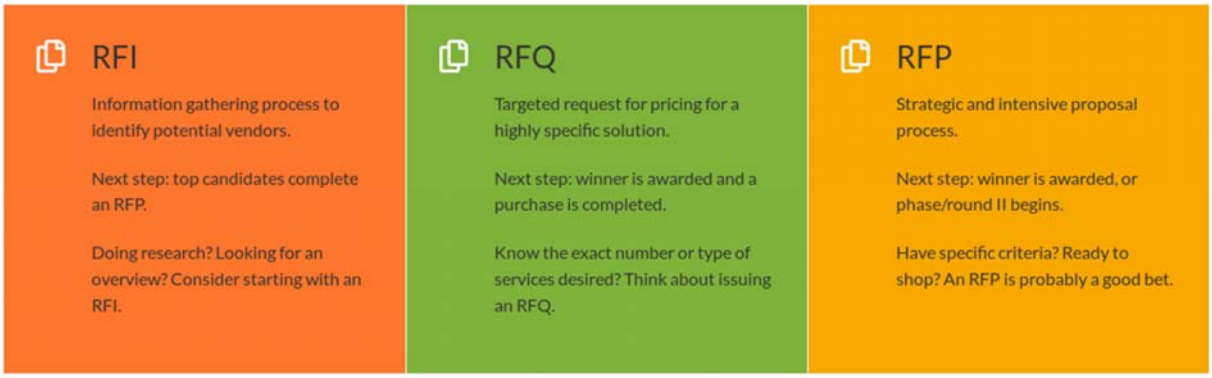
Remember also the fourth quality factor in business: **SERVICE**.

Sw development ”process”

Customer gets an idea for a new information system (or replacing an old one by a new and better).

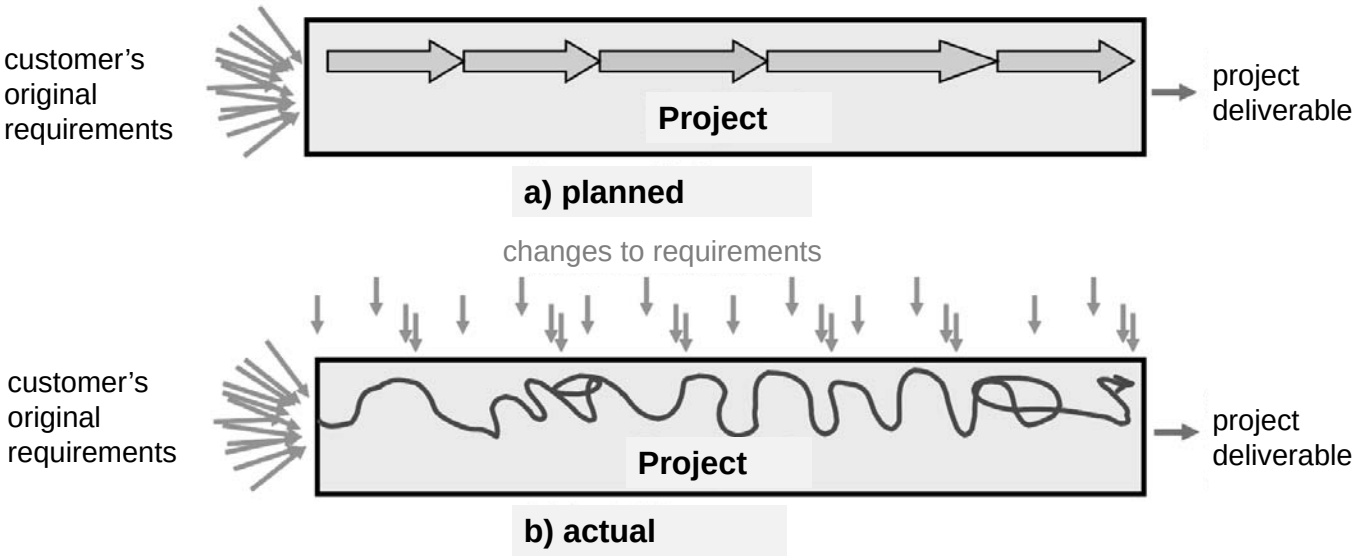
Customer writes down most wanted/needed requirements (functionalities/features), hopefully in a list of priority order (usually in spreadsheet).

If not COTS (commercial off the shelf), customer makes RFI/RFQ/RFP (request for information / quotation / proposal). There are many templates for such document.



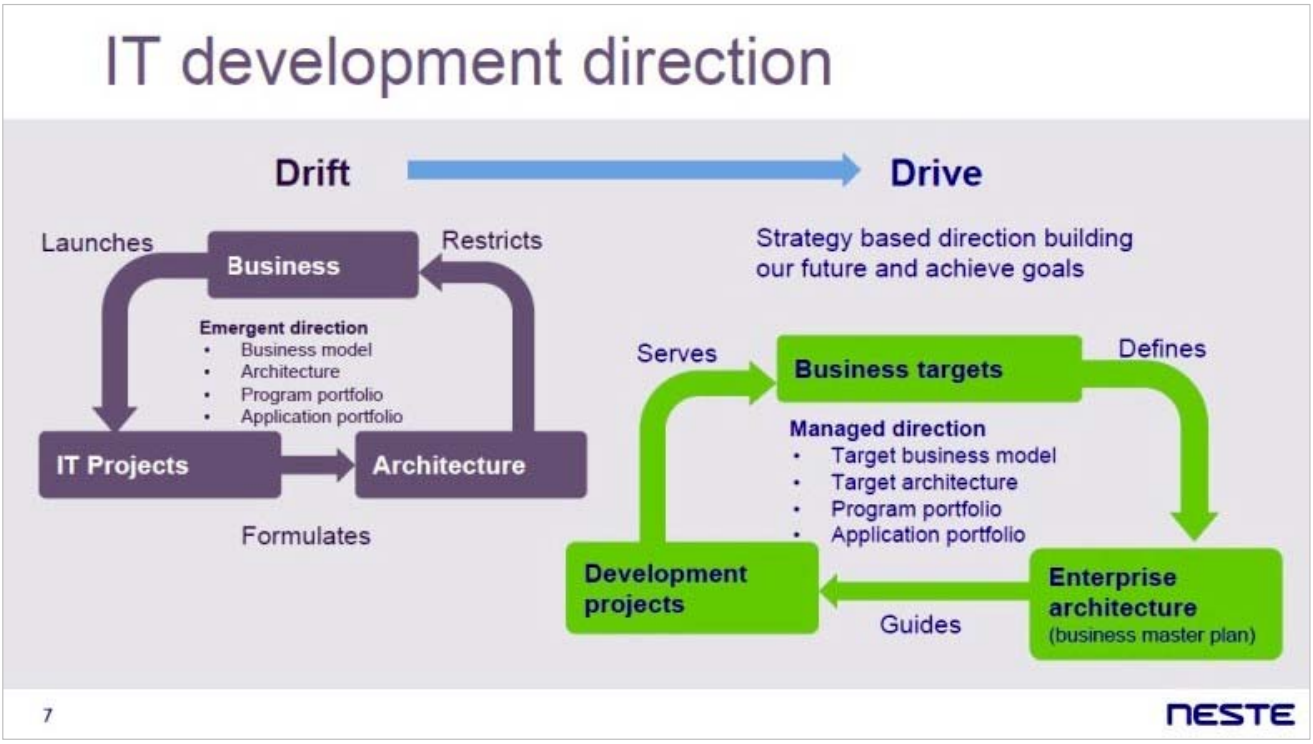
[rfp360.com]

SW project is reacting to changes and (hopefully small) surprises



(compare that to your planned study path…?)

ICT projects should be made to help business



[CIO-webinar 21.11.2018]

ISO 24765:2017 Systems and software engineering — Vocabulary

3.1582 feature

- ***1.*** distinguishing characteristic of a system item
- ***2.*** functional or non-functional distinguishing characteristic of a system, often an enhancement to an existing system
- ***3.*** abstract functional characteristic of a system of interest that end-users and other stakeholders can understand

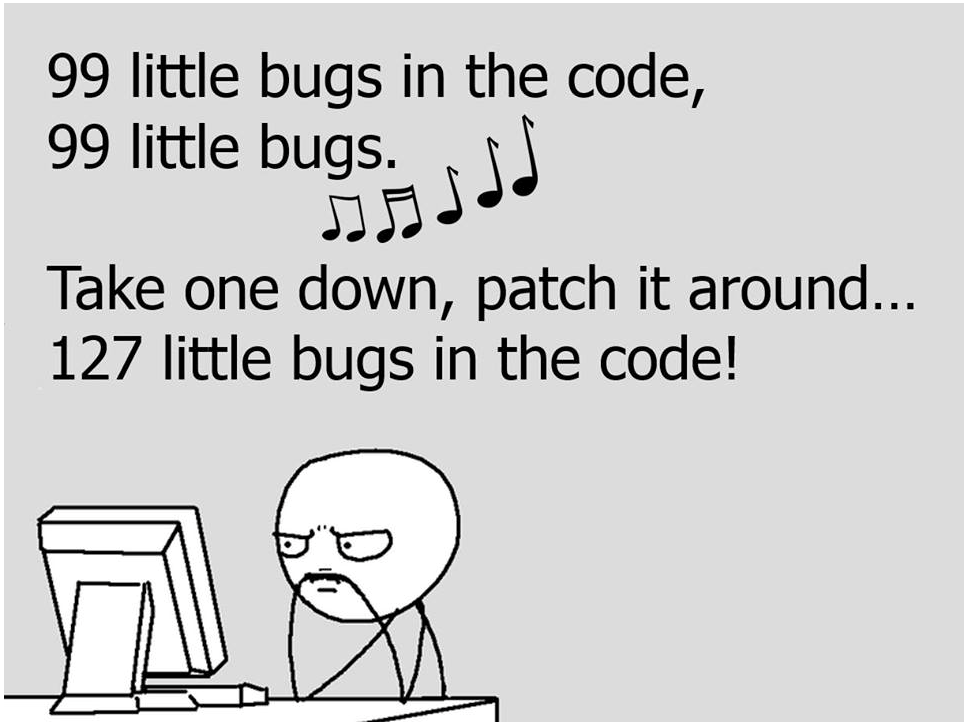
3.1716 functionality

- ***1.*** capabilities of the various computational, user interface, input, output, data management, and other features provided by a product
- Note 1 to entry: This characteristic is concerned with what the software does to fulfill needs. The software quality characteristic functionality can be used to specify or evaluate the suitability, accuracy, interoperability, security, and compliance of a function.

Tampereen yliopisto
Tampere University

Sometimes, especially in old legacy code which has very little if any documentation and very few comment lines, it may happen that fixing one bug causes several other bugs as a **side effect**.

Maintaining old ”spaghetti” code could be a nightmare. **In many cases it would be better to start making a new systems from scratch.**



Tampereen yliopisto
Tampere University

Some other points about sw projects

- Sw projects are not scalabe; what works in five person project, might not work in 50 person project. If one developer/coder produces 100 LOC in one day, 10 developers do not necessarily product 1 KLOC of working and tested code.
- In larger groups/teams, work division and **communication takes more and more time**. Even at TIE-PROJ groups of seven students complain about difficulty to agree meeting times, to get everybody around the same table (F-2-F).
- programming language should be selected according to project characters, as well as the methods used in the project (one method is not good for all project types)
- sometimes customer suggests/requires some programming language and/or method to be used, such cases should be discussed in more detail (are those wishes based on real facts or experience, or just hype).

Tampereen yliopisto
Tampere University

This can only happen in Sw Eng

Let there be a 15 week long sw project (estimated).

- at week 7 swengineers say it is 40 % ready
- w9; 50 % ready
- w10; 60 %
- w11; 70 %
- w12; 80 %
- w13; 85 %
- w14; 90 %
- w15; 92 %
- w16; 93 %
- w17; 93,5 % ready. **Nobody knows what that means ?? Anything working ??**

This is a horror story about an old poor sw project.

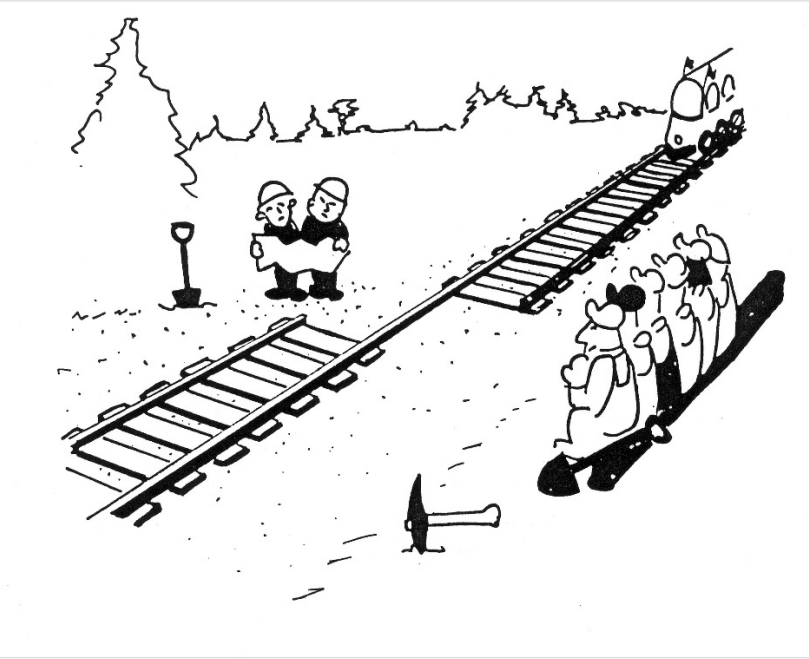
Better than percentage, use features / functionalities to measure readiness.

Tampereen yliopisto
Tampere University

nice planning... but not enough

For understanding requirements right, it would be good to show current demos (GUI or functioning skeleton program) to customer every now and then (every 2..4 weeks), to get feedback.

A Finnish study from large-scale international project revealed that **unofficial communication between vendor and customer** (in seminars etc.) was of great help.



Tampereen yliopisto
Tampere University

Single source = ”vendor lock”

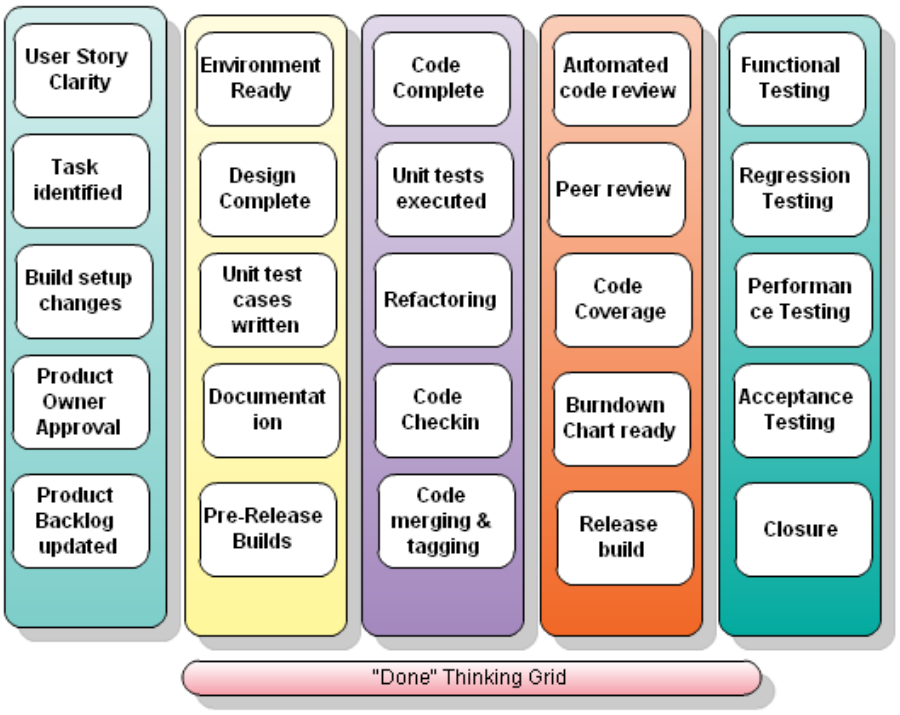
Tampereen yliopisto
Tampere University

Some documentation is always needed in commercial sw projects. E.g. for maintenance work.

Technical debt; e.g. you have made some ”shortcuts”, perhaps some nice ”hacks” to deliver product quickly, but later at maintenace phase that causes troubles.

Tampereen yliopisto
Tampere University

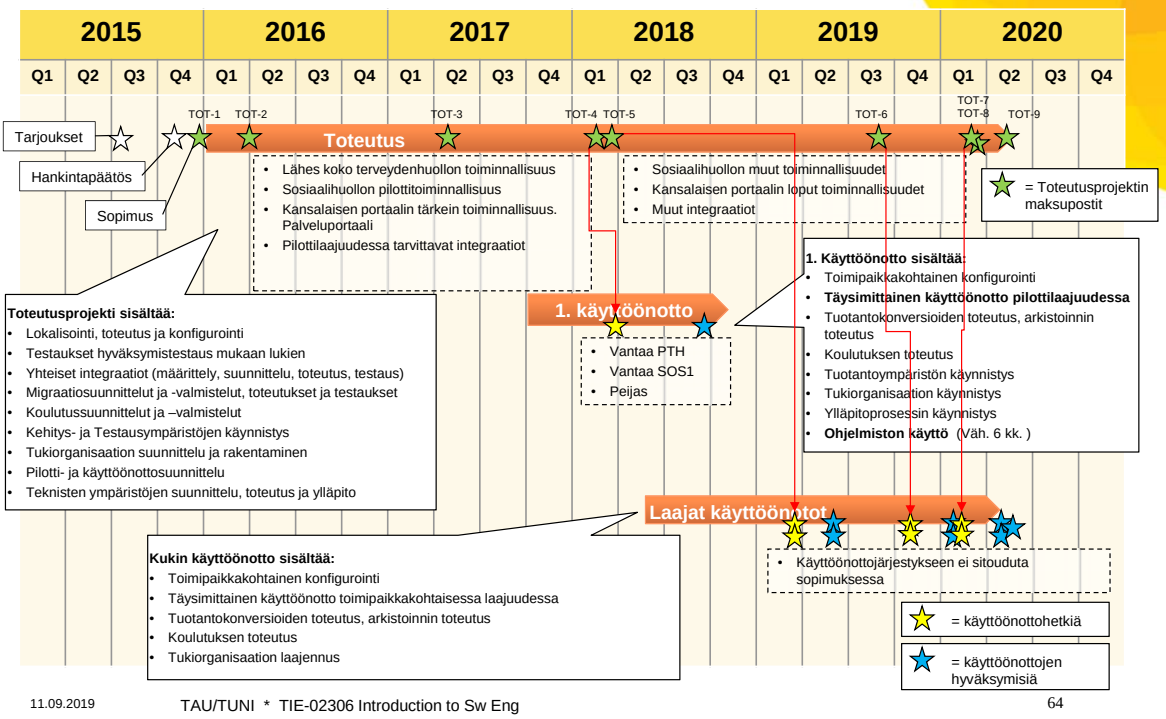
A general definition of done (DoD) can be spelled out in a “done thinking grid”



[www.scrumalliance.org]

APOTTI

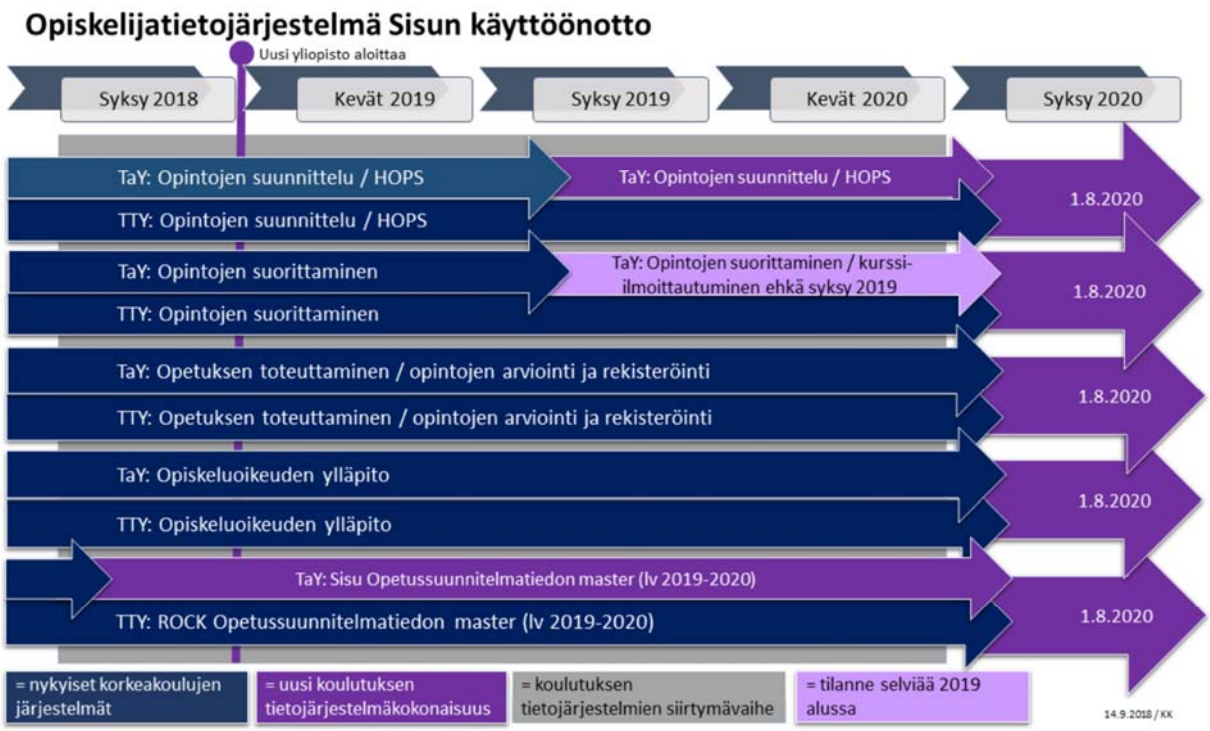
Toteutus- ja käyttöönotto – Pääprojektit ja aikatauluarvio (päivitetty tarjouksen perusteella)



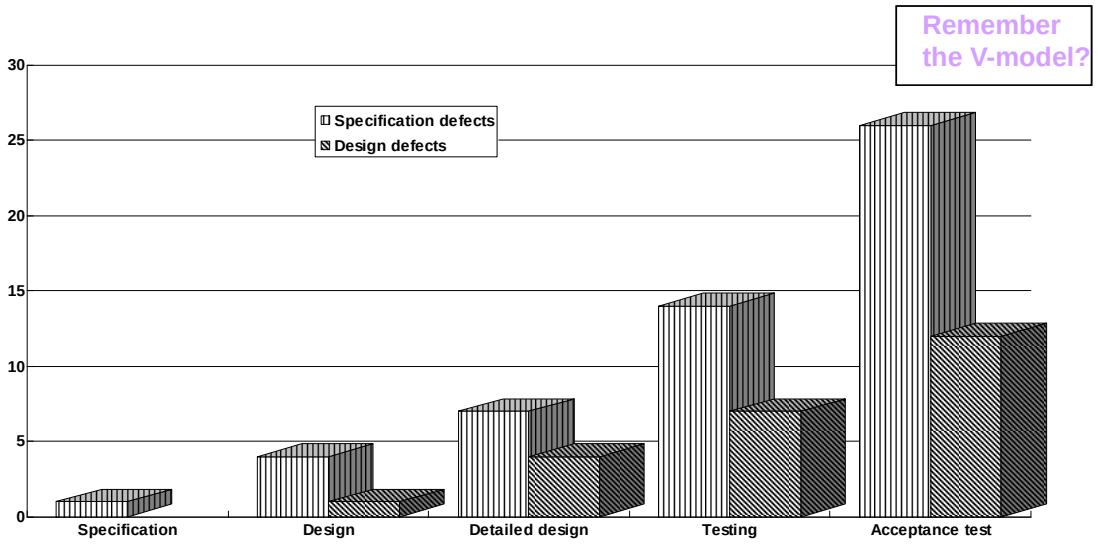
Tampereen yliopisto
Tampere University

https://www.apotti.fi/en/project-follow-up/

Tampereen yliopisto
Tampere University



Relative cost of repairing a defect



[Jalote: an Interactive Approach to Software Engineering, 1991]

Tampereen yliopisto
Tampere University

Ethics, Code of Conduct

IEEE = Institute of Electronics and Electrical Engineers [I-triple-E]

IEEE CODE OF CONDUCT

Approved by the IEEE Board of Directors, June 2014

1. Be respectful of others
2. Treat people fairly
3. Avoid injuring others, their property, reputation or employment
4. Refrain from retaliation (= no revenge)
5. Comply with applicable laws in all countries where IEEE does business and with the IEEE policies and procedures.

The IEEE Code of Conduct describes IEEE members' and staff's commitment to the highest standards of integrity, responsible behavior, and ethical and professional conduct.

Ethics, Code of Conduct

Simplified ethics

- **take care of yourself** – heroic effort as a ”suicide coder” is no good in long run
- respect yourself
- respect (be polite to) your fellow workers
- respect developer/vendor company
- respect customers and customer company
- many problems and conflicts can be solved by discussion
- if you see/discover some error in project, inform others about it (do not hide)
- develop yourself, continuous life-long learning (you can not avoid that).

IEEE Code of Ethics *[https://www.computer.org/education/code-of-ethics]*

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. **PUBLIC** – Software engineers shall act consistently with the public interest.
2. **CLIENT AND EMPLOYER** – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **PRODUCT** – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **JUDGMENT** – Software engineers shall maintain integrity and independence in their professional judgment.
5. **MANAGEMENT** – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. **PROFESSION** – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. **COLLEAGUES** – Software engineers shall be fair to and supportive of their colleagues.
8. **SELF** – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Tampereen yliopisto
Tampere University

IEEE Code of Ethics

1. to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, and to disclose promptly factors that might endanger the public or the environment;

2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;

3. to be honest and realistic in stating claims or estimates based on available data;

4. to reject bribery in all its forms;

5. to improve the understanding by individuals and society of the capabilities and societal implications of conventional and emerging technologies, including intelligent systems;

6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;

7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;

8. to treat fairly all persons and to not engage in acts of discrimination based on race, religion, gender, disability, age, national origin, sexual orientation, gender identity, or gender expression;

9. to avoid injuring others, their property, reputation, or employment by false or malicious action;

10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

Tampereen yliopisto
Tampere University

Tampereen yliopisto
Tampere University

Documentation (don't be scared)

- Every serious software has some documentation.
- **But the amount of documentation should be kept reasonable**, that means minimum… short.
- Good documentation is complete and unambiguous but short. **Have you seen any ?**
- At least users need some guide/manual (at least on-line help (F1), or ”readme.txt” or ”help text” file).
- Some kind of maintenance guide/manual would help developers make later fixes/corrections, adding or deleting parts or functionalities. Important document !
- In longer projects weekly or bi-weekly reports give good information to stakeholders.

Tampereen yliopisto
Tampere University

The long path of documentation may be:

- Feasibility Study / Preliminary Analysis (is the project worth starting ?)
- Project Plan (updating depends of use and purpose)
- Requirements (SRS = Software Requirements Spefication)
- Design (architecture, detailed/module)
- GUI sketches/designs (Graphical User Interface)
- Test Plan (module, integration, system, acceptance)
- code comments (by Coding Conventions / Style Guide)
- User Manual / User Guide
- Maintenance Guide (for developer company)
- Test Report (system, acceptance)
- Project Final Report / Lessons learnt.

Not all these are made nowadays in the same project; need for documentation depends of the project and customer.

Tampereen yliopisto
Tampere University

reuse, there could be more of that

Why ”invent a wheel”, if it has already been done earlier ?

Reuse is something that is **not yet fully utilised** at software industry.

Ideally, some 50 % of code could be reused, but the actual amount is something like 10..15 %. Well, of course if you are making similar product to several customers, you may reuse even more than 50 % of the code (which is reasonable).

Two major blocks for reuse are thinking or acting in a wrong way;

- ”Nobody has done this before, it takes me just a few days to code…”
- ”Somebody may have done this earlier, but where the code is and how can I find it… and if I find it, how to find out how it works…”

On the other hand, if you reuse working code, you still need to test it in the new product !

Tampereen yliopisto
Tampere University

Tampereen yliopisto
Tampere University

Highlights - What to remember

- at every process model there is some amount of: **requirements – design-implementation – testing**
- today's aeroplanes and even cars have millions lines (MLOC) of code
- software projects are not scalable; what is OK for small project, may not succeed in large project
-
-
-